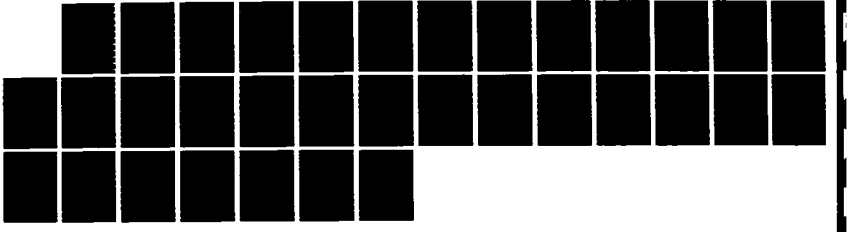
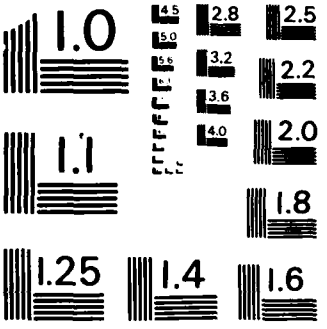
AD-A169 759 E-L DESIGN RATIONALE (VERSION 00)(U) SOFTWARE OPTIONS 1/1
INC CAMBRIDGE MA 15 MAR 86 N00014-85-C-0710

UNCLASSIFIED F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART

NATIONAL BUREAU OF STANDARDS - 1963 - A

E-L

Design Rationale

(Version 0.0)

Contract No. N00014-85-C-0710

March 15, 1986

Software Options, Inc.
22 Hilliard Street
Cambridge, Mass 02138

E-L

Design Rationale

(Version 0.0)

March 15, 1986

Software Options, Inc.
22 Hilliard Street
Cambridge, Mass 02138

Table of Contents

1. Introduction

One of the great contributions of Ada to the field of programming languages is the notion that a language should have a design rationale document. One always wonders, confronted with some bizarre aspect of a language, whether it is deliberate or accidental. A rationale, more by omission than commission, can reveal the answers to such questions. It is thus with a deep sense of duty, and no small amount of trepidation, that we offer this document to the curious user of E-L.

The Ada rationale is organized along the structure of the language—the reason for this or that construct is such and such. In discussing the design of E-L, there are a number of broad issues that permeate the design, and would not fit such an outline. Thus this document is a collection of relatively independent essays that address what we felt to be important issues in this design.

2. The Philosophy Behind E-L

E-L is a software development environment designed to achieve a dramatic increase in software productivity while retaining or improving the efficiency of software produced. To appreciate the rationale for its design, one must first understand in general terms where there are productivity gains to be made by using a new kind of software development environment and what steps are needed to ensure that product efficiency will not suffer.

There are three main ways in which E-L will improve productivity by comparison with current practice. The first is by supporting software development and evolution as an iterative, exploratory, on-going process, rather than as a straight-line sequence of steps. The second is by aiding the re-use of design and implementation knowledge. The third is by permitting programs to be written at a level and in a style of language that is most appropriate for the application at hand. Let us briefly discuss each of these aspects of E-L.

We begin by observing that the cost of *maintenance* usually dominates the cost of initial development in the construction of a large, long-lived software system. The word "maintenance", of course, is a widely-used misnomer that encompasses not only "repair" but "enhancement" and "evolution". A large software system is not static like a piece of machinery. Software can rarely be assembled once from pre-specified parts and then remain rigidly the same for its useful lifetime. It is more like a living thing that begins to function early in its development and then grows gradually into maturity. Like living things, software systems must be adaptable to changing stimuli, i.e., to demands for new behavior, new functionality. Therefore the product of a software project should not be just some code, a user's manual and a service manual, as if the program were a machine. The product should include the wherewithal to enable a "maintainer" (i.e., a re-developer) to *resume* development by altering specifications design, code and/or documentation. He should have should have access to

- up-to-date specifications for the system,
- a documented development plan (a "program-in-the-large", showing how system components are put together and what resources they provide one another),
- a family tree, if the system is being maintained as a multi-version family,
- prototype models used for testing particular aspects of the design,
- facilities and data for regression testing,
- instrumentation data gathered in previous use of the software,

and whatever other materials an initial developer would have needed. Obviously, these items are worse than useless if they are not kept up-to-date. If specifications and documentation fall out of correspondence with the implementation, software becomes brittle and eventually unmaintainable.

E-L will enhance productivity, therefore, by reflecting the view that software development is in general an iterative process in which specification and documentation are intertwined with implementation. It will support a software database containing not just code and manuals, but all the documents and data needed to resume and revise development. The database will explicate the interrelationships between these items, e.g., by linking documentation and code at a detailed level. To the greatest extent possible, items in the development database will be usable by tools; for instance, a development history will be "replayable", so that software can automatically be regenerated after a modification. In short, E-L will make it much easier for developers to invest in supporting the future enhancement of their products, rather than treating software construction as an assembly-line operation.

A second way to increase productivity is to support re-use of proven design and implementation knowledge. A great deal of software engineering amounts to innovative application

of well-understood algorithms and data structure classes. It ought to be possible to reduce the cost of adapting and coding these algorithms and data types. Unfortunately, the use of library packages to avoid this redundant effort has not borne much fruit. It is often not sufficient to tailor a package by setting some parameters, either because an application lies outside the capabilities of the package or because efficiency considerations dictate optimizations that depend on subtle facts about the setting in which the package is being applied. E-L supports a technique for encoding and re-using knowledge about how to apply generally useful algorithms and data structures and how to tailor combinations of them to operate efficiently in specific circumstances.

A third way to increase productivity is to arrange for programs to be written at a level and in a style of language that is appropriate for the problem being solved. It is generally agreed that software productivity can be increased by raising the level of the language in which programs are written. This has been borne out by studies comparing projects implemented in assembly language with others using high level languages. As the level of abstraction becomes higher, programs become more concise, more easily understood, easier to change, and more machine-independent. We can expect continued improvements in these areas if we can lift the programming level even higher. In fact, if performance were no object, it would be relatively easy to produce functionally correct software more economically.

Unfortunately, given current compiling technology, raising the programming level usually causes the efficiency of the final product to drop. A guiding principle behind the design of E-L is that *efficiency matters*. Our aim is to support development of large, long-lived production software systems in which the cost of execution is an important figure of merit; sizable embedded systems fall into this category, as does most system software (e.g., development environments themselves). In this respect, E-L differs from other highly integrated environments, such as Smalltalk, which give strong support to rapid development of well-structured programs but provide only for relatively inefficient execution, entirely within the development environment. E-L is designed to allow production programs to be extracted completely from the environment. Our plan is ultimately to use the method of highly optimized code generation that we are now developing (cf. [CGBC]) to make the programs produced as efficient as could be produced by hand or more so. There is no doubt that the attainment of acceptable efficiency is one of the chief sources of complexity in programming, and there is good reason to believe that the need for human insight in making certain programs efficient will extend many years in to the future. However, it is our thesis that by acknowledging the efficiency problem, and by giving programmers both high level means of specifying algorithms and *separate* ways of obtaining efficiency, we can truly raise the level at which programming is done.

To achieve a clear programming style and an iterative, exploratory development style, re-using design and coding knowledge while still producing efficient software, we use a method that we call *transformational refinement (TR)*. We will say more about this method in subsection 1.1; for now, it suffices to mention that TR is most successful when used with a *wide spectrum* programming language and that it depends on the support of efficient tools for mechanical application of transformations and for semantic analysis of programs. A wide spectrum language is one in which many styles of programming, ranging from specification-oriented to machine-oriented, can be accommodated within a single language framework. Before beginning the design of E-L, we had experimented with TR for several years using a programming language that had not been designed with transformational refinement in mind. That experience contributed to our decision to create simultaneously a new development system that would support TR *and* a new programming language that would cater to the needs of the environment and would support the transformational methodology. The name *E-L* stands for *environment and language*. E-L is one of very few development environments in which the design of the programming language has been made subordinate to the needs of the larger system, and it is the only such environment we are aware of that is intended to produce efficient, free-standing production programs.

The design principles behind E-L follow directly from the methodology that it is intended to support. A true wide spectrum language will either be based on strong facilities for extension by its users or else it will be a "kitchen sink" language, incorporating an irregular jumble of features intended to accommodate all styles of programming. E-L provides for extension of syntactic notations, of data types, and of control structures. On the other hand, extension facilities must not

be used in an undisciplined manner. The importance of uniformity and of being able to analyze programs at all levels and in all styles follows not only from the needs of human readers of code, but also from the needs of tools that use analysis in supporting program realization. Therefore, the extension facilities of the system are governed by axiomatic constraints that ensure regularity and enable tools to depend on assumptions that they could not always detect or verify for themselves. The language has a firm semantic basis in that most of its built-in facilities are reducible to a small, uniform kernel language. In fact, the reduction mechanism used to describe the language rigorously will be *identical* to extension facilities that are provided for users. Thus, the formal definition of the language is unusually accessible to its users and can serve both as an explication of built-in features and as a guide for user extensions. Finally, the desire to enhance productivity and reliability by re-using programming has been a strong motivation, particularly in the design of the function family and data type facilities of the language.

Although E-L is evolutionary in the sense that it embodies research that has evolved gradually over recent years, we view it as a revolutionary step because we have chosen not to constrain the design to be compatible with conventional practice, conventional formal semantics, conventional host equipment, or conventional syntax. While we have not been whimsical in choosing to break conventions, we have felt free to make choices that can lead to more economical program development, even if those choices imply significant dependence on modern workstation technology and some investment in learning a new language and a novel style of software development.

The following subsections expand on the philosophy behind E-L by summarizing transformational refinement and its implications for language design.

2.1 Transformational refinement

Transformational refinement (TR) is an approach to the process of implementation that we have been studying and experimenting with for the last several years. It reflects our view that, to the greatest extent possible, implementations should be derived mechanically from their specifications. Our particular focus is on the derivation of efficient, readily compilable code from a generic program model that represents the design of a whole family of implementations. We strive to formalize and mechanize the *process* of software design and implementation, so that the rationale behind this process is clearly exposed and documented, and so that major parts of the design can be re-used when a software product needs to be adapted to a new hardware configuration or to new requirements for performance or functionality.

Under TR, the system implementor's task is not just to produce a final, concrete system that matches stated requirements. Rather, it is to produce a very high level program, which we call a *program model*, together with one or more formal *plans* for its realization as a concrete system. A plan is a structured description of the mechanical steps needed to refine the program model by applying program transformations. The transformations may either be developed by the implementor or they may be drawn from a collection of rules whose validity has been certified and whose domains of effectiveness are well documented. The program model, the development plan, and the project-specific transformations are actively maintained for the life of the software system. They are the keys to understanding what the system does and how it works.

Transformational refinement increases productivity and reliability without diminishing the efficiency of systems that can be produced. It enhances productivity both because coding takes place at a high level of abstraction and because the method promotes the re-use of design and implementation knowledge. Program models from which implementations are derived can be concise and clear because they omit implementation-dependent details and they employ notations and abstractions tailored to the application domain. Deriving concrete programs mechanically instead of writing them by hand facilitates re-use of previous work not only because the method encourages program modules to be written in a highly generic style, but also because so much knowledge about the design and realization of each system product becomes formally encoded and documented and remains available for re-use, either in the evolution of that system or in the development of completely new ones.

Our methodology promotes reliability because the correctness of a high level program model written in domain-specific terms is more easily checked—whether by inspection, test execution, or formal verification—than that of a low level implementation, and because transformation rules can be independently verified prior to system derivation and checked for applicability as they are used. The method permits highly efficient software to be developed in spite of the use of a high level coding style because the implementor's insight can enter into the selection of data representations, access methods, low level communication protocols, and other details that are appropriate to the target configuration and to performance requirements.

The successful practice of transformational refinement for developing large software systems depends critically on computer support, both because transformation of large amounts of code by hand is unacceptably tedious and error-prone, and because the method entails retaining and re-using a great deal of information about each software project. It demands effective tools for applying transformations and for performing the semantic analysis necessary to guide and validate them. It requires a powerful and capacious project database to hold the program model for a software system, the concrete instances derived from it, and analysis for all of these versions, together with enough dependence information to permit the developing system to evolve incrementally with a minium of rundundant effort. A shift towards more machine-intensive methods in order to raise productivity makes sense economically, of course, since skilled personnel become ever more expensive while the cost per unit performance of computing capacity continues to drop. E-L's design, while not dependent on exotic machine architecture, does assume the use of modern workstations with high bandwidth user interface equipment and considerable processing power and storage.

The design of tools for program transformation and refinement is an active area of research, by us and by others, and does not fall within the scope of the E-L project. Nevertheless, our

experience with TR to date has strongly influenced the design of the environment and the language. The decision to provide a robust and extensible facility for attaching annotations to programs down to the expression level, for example, will support TR both by allowing the attachment of analysis results directly to code and by providing direct links from a program model to the concrete code that realizes it. Also to facilitate transformation, we designed the language to accommodate a wide spectrum of programming styles within a single framework. This aspect of the design is the subject of the next section.

2.2. Wide spectrum extensible language

In one sense, transformational refinement is an extension of the familiar process of program compilation, differing in that it allows a software developer to use his insight to control detailed choices about data representations and the optimization of high level algorithms. However, TR is different from compilation in several ways, and the special requirements of the transformational approach have influenced the design of E-L. Whereas compilation is the simultaneous translation of all aspects of a program from one language to a very different one, refinement is an iterative, incremental process in which each step may realize only a single aspect of the program. The reason TR proceeds incrementally is that users must be able to understand and to compose individual transformation rules. A rule drawn from a standard library must have a narrow, clearly identified purpose so that it is easy to combine with others. A rule provided by the user should not be forced to accomplish many tasks at once; if rules are complex, then the structure of the implementation will be obscure. A transformation may introduce lower level language features than those used in the version of the program being refined. It may be part of a series of transformations designed to reduce the program entirely to low level terms. Nevertheless, the program that results from applying a single transformation must not be in semantic limbo. The newly inserted pieces must fit together meaningfully with the parts left unchanged, because in order for refinement to continue, tha analysis of the new parts must be integrated with the analysis of the old.

Most languages don't offer a wide enough range of styles and abstraction levels. E-L is designed to embed a wide spectrum of language styles within a single framework. In principle, the spectrum of language dialects that could usefully be incorporated in a transformational development environment is very wide indeed, ranging from high level non-procedural specifications on the one hand to silicon chip designs on the other. These extremes are outside the scope of the E-L project. Our objective is to make it possible within E-L to create a clear procedural model of a software system or family and then to refine that model to the level at which an optimizing compiler can produce a highly efficient production version of it. At the model level, the programmer must feel free of language constraints aimed at promoting efficiency and must be able express his algorithms in terms natural to the problem domain. At the compiler input level, the mapping from code to machine facilities must be direct, so that it is clear which parts of a program are expensive and so that optimizations beyond the capabilities of the compiler can be made explicit by the developer.

A couple of examples may illustrate the ways in which E-L satisfies the need for a wide spectrum language. At the model level one can use the type mechanisms for behavioral effect and for modularization without being forced to deal with constraints intended only for efficiency. The fundamental built-in numeric types, such as `integer` and `rational`, are abstract types describing ideal numbers; constraints resulting from the finiteness of computing machines are introduced by refinement of these abstract types to machine-oriented realizations (which will typically be built-in themselves). E-L's facilities for polymorphic programming are extraordinarily flexible. Parameterized types are user-programmable, finite and infinite union types are supported, and the parameters of functions can be described by type patterns or "templates", rather than specific types, to maximize the re-usability of function definitions. On the other hand, all of the necessary declarative mechanisms for refining programs to strongly-typed form are included in the language. Furthermore, the axioms for the behavior of type operators inhibit undisciplined or counter-intuitive uses of type flexibility.

Another aspect of programming that illustrates the wide spectrum approach is storage management. A program model should deal with the behavior of values and objects without concerning itself with managing the storage they occupy. Efficient storage handling can be introduced during refinement by making use of facilities (pointer types, heaps, and so on) that are not normally needed at the model level. Moreover, because refinement is based on analysis of a particular program, this approach can produce much more efficient and/or reliable results than would be possible with a language that predefines a fixed set of storage management facilities. For instance, programmed deallocation of storage, which is ordinarily excluded from high level languages and is a source of hard-to-detect bugs in low level programs, can be used safely if it is introduced only by correct transformations.

It would be disastrous to design a wide spectrum language as a large monolith. It would be

impossible to anticipate all the facilities that would be needed, either at the program model end of the spectrum or at the machine-oriented end. And the result would almost certainly be a semantic jumble, filled with unexpected interactions and ambiguities. Therefore, E-L is an *extensible* language system. While the built-in langauge provides an ample basis for programming in a number of styles, it will sometimes be inadequate. In that case, nearly every aspect of the language, including syntax, data types, function families, declarative constructs and control structures, can be extended by users. In fact, most of the built-in language is defined formally from a very small kernel language using exactly these facilities. (Thus, experienced users of the language will be able to understand and make use of its formal definition.)

Furthermore, E-L's extensibilty does not come at the expense of analyzablity, since that would defeat the purpose. A program model can only be correctly refined if it can be analyzed in sufficient detail to guide program transformation. Therefore, every extension facility is governed by associated axioms that must be obeyed by user-provided extensions. In many cases , adherence to the axioms can be verified; in a few cases, they have the status of unpoliced rules of the game. In any case, tools can assume a high degree of regularity and predictability in user-defined extensions, without having to interpret their implementations in detail.

Some earlier extensible languages have been unsuccessful because they attempted to build on too low level a base language and because they expected every tool cope with any program in the language. Since every abstract construct was defined by reduction to a low level implementation model, tools were not able to make use of subtleties apparent only at the abstract level. In E-L, not every tool deals with the full spectrum of the language. A certain analysis tool may handle only the model-oriented end. A code generator may only handle the machine-oriented end. There may be different transformation tools for different phases of refinement.

The built-in tools accommodate the built-in language styles, together with modest extensions. For more elaborate extensions, it may be necessary to extend the set of tools by adding new ones. For example, it might be desirable to add a logic programming style for use in program specification. If so, one could make specifications executable by adding a resolution-based logic language evaluator to the E-L tool set. At the other end of the spectrum, an unusual target machine architecture might occasion the need for special machine-oriented language facilities and for a special code generation tool to compile them.

In short, E-L is able to cover a wide spectrum of language styles not only because of its language extension mechanisms, but also because its tool set is extensible. The environment is conducive to tool extensions like those mentioned above because of the generality and flexibility of its project database and its program annotation facility.

3. State

3.1 Machines and logic

Turing's machine and Church's lambda calculus have served as intellectual tools for computer scientists since before computers. The former has been used chiefly to study the complexity of computational problems, including undecidability, completeness, and related issues. If we expand the notion of machine to include finite state and pushdown automata, multi-tape and random access machines, and a few other variants, we have the basis for most known and conjectured results for upper and lower complexity bounds.

The lambda calculus has had a somewhat different role. While the computational power of the lambda calculus was shown early on to be equivalent to a Turing machine, the lambda calculus has served a role in serious language designs beginning with Algol [Algol 58, Algol 60]. If we expand the notion of "the" lambda calculus to include variants in which lambda expressions, i.e., functions, can be named (to allow an intuitive notation for recursion), it provides an elegant linguistic mechanism for the description for many computational problems. Indeed, it is our view that the important contribution of the lambda calculus to practical programming is not its relation to formal semantics—most programmers don't care about formal semantics anyway—but rather it use as an organizational tool in expressing programs. There is no doubt that for most purposes it is far more appropriate than a finite state machine cum memory model.

These intellectual traditions would not have quite the appeal they do without the existence of hardware. Hardware works by changing state. This makes hardware irrevocably more closely tied to the mathematical "machines" discussed above. This is not a coincidence, of course, for these mathematical ideas were created and named with physical devices in mind. But the existence of hardware leads to the software problem, and utilizing these devices is best done with languages that are largely inspired by the lambda calculus, and there is little argument that the more heavily a language relies on a stateless view of the world, the less efficient it will be. This is only fair: if you don't pay attention to matters of efficiency, you will have less of it.

A language design picks some way to wed state and the lambda calculus. Sometimes, this is to the exclusion of lambda calculus, as with machine language or BASIC, two languages noted for their lack of suitability in large programming problems. Other languages combine the two by the device of writing in something that resembles lambda calculus, but then allowing "assignment" to the formal parameters of the lambda expressions, thereby changing the values that they previous had, and maybe or maybe not changing values of other parameters, by now typically known as variables. Languages that use this approach include Pascal and current LISPs. Other languages. the "applicative" or "logic" languages, attempt to abandon state entirely. This has lately achieved considerable academic respectability, but it is not a satisfactory basis for a wide-spectrum language. On efficiency grounds alone, it would have to be discarded, for efficiency is evidently not easily achieved from these starting points [SETL]. But there is a further problem, in that this view of computing is not sufficently rich. There are many computing problems, from numerical simulations of physical systems, to the management of windows on a workstation screen, in which "state" is a natural part of the *problem*, not only an artifact of the computational solution of a problem. The stateless view of computing isolates the programmmer from the natural formulation of the problem. The necessity to achieve efficiency or deal with problems involving state often causes the modification of stateless languages with features that let notions of state creep back in, such as the "prog" feature in LISP (once an applicative language).

The fundamental semantics of E-L fully embraces the notion of state, without apology or excuse. It also embraces a very lambda-calculus-like kernel representation and interpretation of programs. The combination is not done by compromising the lambda calculus with allowing the assignment of values to formals, but in a quite different way, which we hope is a happier synthesis of the two traditions. This approach has consequences that percolate up to the surface of the language, and our present purpose is to show how some of the more unusual aspects of E-L originate from this semantic basis. The essence of E-L's connection of machines with the lambda calculus is that there are certain *values with state*. There is a sharp distinction between a value with state, call it v, and the value or values that one imagines describing the state of v. In subsequent

sections, we shall review the principal values with state that are built-in to E-L.

.2 Places

The paradigm for values with state are place values. For example, suppose that *v* is a place value, and that *w* is another value that currently describes the state of *v*. It is highly unikely that $v = w$; in machine terms, this would correspond to location 211 containing the value 211—not impossible, to be sure, but only coincidence, and probably a fleeting one at that. Of course, if the type of *v* were `place(integer)`, then it would be impossible for its contents to be itself, because the contents would have type `integer`, which is not convertible to a `place(integer)`. Many languages get by without ever making `place(integer)` a notion that the programmer must think about. Considerable insight in to the reason for the elevation of this notion into the (at least occasionally) visible part of the language comes from recalling the semantics for `assign`, using Floyd-Hoare notation for pre- and post-conditions:

ASGN-CNT1 For a place *p* and a value *v*:

`assign(`*p*`,` *v*`)`
`{contents(`*p*`) =` *v*`}`

ASGN-CNT2 For places p_1 and p_2, and any value *v*:

`{contents(`p_1`) =` *v* and $p_1 \neq p_2$`}`
`assign(`p_2`,` *v*`)`
`{contents(`p_1`) =` *v*`}`

The crucial aspect of these rules that distinguishes them from similar rules in other languages is that the equality function (= and #) that is being talked about is the equality of E-L itself. In particular, the question of whether or not the places themselves are equal ($p_1 \neq p_2$) is very often a meta-question in semantics of other languages.

There is a good reason why the equality of places should not be a meta-question: the need to simultaneously have both analysis and extension. For example, determination of common subexpressions relies heavily upon being able to ascertain the limits of the side effect of an assignment—it relies heavily on ASGN-CNT2. In order for analysis to be possible, then, there must be some such rule for the behavior of `assign` and `contents`. On the other hand, extensions like the sparse array extension (RM13.5) rely upon being able to extend exactly those two functions. Indeed, an extension of this kind is impossible in any other language known to us. Now, if the rule governing assignment is a meta-rule, relying on a model of the program outside the program, then the possibility of extensions is ruled out. If the extensions are ungoverned by any rules, then analysis is ruled out. To provide both, we have stated the rules for `assign` and `contents` in terms of a function which is itself subject to extension, namely, =. It might have been more appropriate to call the rules ASGN-CNT-EQL 1-2, because they actually state relationships between all three functions.

Let us consider a program in kernel E-L, i.e., one in which reductions do nothing. Its evaluation is specified by a canonical tree walk of the program, with functions applied in the usual way introduced to the computing community by Algol. If the program has no values with state, then this algorithm gives it the usual denotational semantics. However, if the program does have values with state, then the order of the scan matters. The values with state are produced, ultimately from values without state, by primitive functions like `new-place`. Once such a value is produced, it is passed around like other values in the function application process. Occasionally, however, there are applications of other primitive functions, like `assign` or `contents`, that change or query the state of the value with state. Sometimes, the answers from these queries affect

the flow of control of the walk that describes the program's semantics. In essence, the query functions act like oracles. In values like places, these oracles are controlled by the program; in values like input ports, the oracles are uncontrolled. And dually, some changes to state may change later behavior of the program, as in places, and others do not, as in output ports.

3.3 The limits of place

The question arise, is perhaps the notion of place not sufficient? Can't we always view any value with state as the place of some sufficiently complicated value that does not have state? The answer to these questions is yes, from a sufficiently theoretical point of view, but a definite no from the point of view of axiomatic style and mental model. In many cases, it is not natural to retrieve the entire state of a value with state using a single function application. For a place this is natural, because intuitively it holds only one value. But for a table, it is more natural to describe the behavior of, and program in terms of, the `enter` and `lookup` functions. One could formalize a table as a set of key-datum pairs, and could of course axiomatize it in this way. But this is not the way one really thinks of a table, and certainly not the way it is implemented. It is just as simple to axiomatize the behavior of `enter` and `lookup` in the style of ASGN-CNT 1-2, and in this style, one never talks about the whole state of the table, only about incremental changes to it. Even if you want access to all of the entries of the table, say for printing it out, it is more natural and more efficient to provide and use an iterator than a function that produces a giant set of all the key-data pairs. In the table case and in others, it may be logically possible to provide an analogue to `contents`, say by extending it to the type in question, but there are cases where the provider of an abstraction does not want to allow easy access to the entire state. For example, in a database with sensitive information, users might be allowed to make updates, and to ask about certain summary information, like averages, but not be allowed to ask about particular entries.

Further evidence that places alone are not enough is notational. In passing values with state as parameters, one almost never wants what would be potentially massive contents of the place, one wants the place itself. Sometimes this is for reasons of efficiency (not to be ignored!); sometimes it is because the whole point of passing it as a parameter is to update its state—grabbing the entire state, i.e., essentially copying it, would be incorrect. If the only way to describe values with state is as the place of some other value, then every formal that takes such a value needs a **`shared`** bindclass. Such a notational burden is not the end of the world, of course, but it indicates that the concept is not quite right.

Still further justification for allowing values with state that are not technically speaking places arises from a consideration of list structure. For example, a tree that can grow and shrink, as well as have node labels that change, could be formalized as the place of a nested tuple. This is very strained, and a horrible image to implement directly. The problems in thinking this way multiply if one has not merely a tree, but a graph with cycles, whose connectivity varies over time. The semantics that E-L uses are quite direct: a value with state may have constituent values that are themselves values with state. In this example, of course, the image becomes very pointer like. Stated differently, it says that pointers may lead to other pointers, hardly revolutionary.

3.4 The treatment of variables

The reader of the E-L Definition cannot help but notice that a great deal of machinery is brought to bear on the treatment of variables, specifically, in hiding their true value-with-state nature from the casual programmer, i.e., from the surface of the language. If E-L's view of places is so superior to the usual notion of variables in other programming languages, why not just force the programmer to program in terms of places? The answer to this is partly historical and partly notational. There is a long tradition in programming of "variables", of being able to assign to them, and to use their values (contents). Beginning with FORTRAN, one could utter statements like "A = B". To the modern, the use of the equality symbol for this purpose seems particularly unfortunate, and this custom was quickly abandoned—Algol used := instead. But the notation that stuck was that the name of a variable could occur on the two sides of the assignment operator, even though it evidently means quite different things in the two places. The question is not why `assign` should

take arguments of completely different type, but how the custom of writing assignment in this way ever arose in the first place. At this point, one can only speculate, but a possible explanation is that what the early designers had in mind was the machine instruction syntax "LOAD A,B", which copies the contents of location (place) B into the location A. The assignment statement "A=B" was thought of simply as an encodement of this instruction.

There is another possible explanation. Knuth's famous empirical study of FORTRAN programs [Knuth] found that a very large percentage of assignment statements were in fact of the simple form "A=B". This might be an artifact of the programming style that FORTRAN encourages, but it nevertheless should give a language designer pause: the act of setting one variable to the current value of another may be a very common thing to do, and we were reluctant to replace a familiar and comfortable notation with one that was both novel and potentially awkward. This reluctance was strengthened by experience with several languages in which identifiers are generally regarded as places, and the contents operation must always be explicitly indicated. The first of these, to our knowledge, was (B)CPL, and later there was BLISS. In each, the contents operator was indicated by ".", indicating some concern on the designers' part with cluttering of the program text. The consensus is that even the dot is a visual nuisance.

In order to provide both familiar syntactic structure and novel semantic structure, we introduced the notion of mode to mediate between the two worlds. The only part of this seen by the naive user is bindclasses, like **`shared`**, but these are easy to relate to similar markers in other languages, like "var" in Pascal or "in out" in Ada. The slightly more sophisticated user might want to extend `assign` and `contents`, but even this can be done relatively naively—the mechanism connecting `<-` with `assign` need not be fully understood in order to utilize the extension mechanisms. It is not surprising that one of the most commonly used syntactic aspects of the language has the most tortuous connection with the semantics of the language; the same kind of thing happens in natural languages, where the most commonly used verbs tend to be the most irregular. We anticipate that both the syntax of variables and the semantics of places will turn out to be quite comfortable.

3.5 Arrays

In deciding on the precise question of semantics for arrays, the question of whether an array is essentially place-like must be answered. Even if places are not a sufficient basis for all values with state, can't an array be treated as the place of a tuple with a well defined structure? E-L does not treat arrays in this way, but arguments can be made in opposition to this decision, and there are syntactic and "mental model" consequences of the choice, so the various options are considered here in some detail. We begin with a short review of how E-L does treat arrays. We start with the premise that one wants to change the value of individual components of the array. In order to base array semantics as far as possible on semantics of simpler objects for which there is independent need, it seems natural that the fundamental operation on an array be a function that takes as arguments an array and an index—a tuple of integers, to support multi-dimensionality—and produces a place. This function is called `pselect`, axiomatized thus:

PSEL0 The function `pselect` is well-defined, pure, and has two arguments, the first of which is a `compound-place` (of which array is an instance). The result of `pselect` is a place.

PSEL1 The function `pselect` is 1-1 on its second argument.

We take as given that basic extensible families be well-defined. This requires that, at least in the kernel language, that arrays be equal according to whether they provide the same map of indices to places, not according to whether the contents of these places are equal. Arrays work in surface syntax the way they do in kernel semantics. Suppose you define two arrays as follows:

```
Array A(3, 5) initially 0
Array B(3, 5) initially 0
```

If you ask `A = B`, the answer is `false`, because they are different sets of places. It is tempting to think that you might want the answer to this to be `true`, after all, it is evident that `A` and `B` are different arrays, and the question that you are really going to be asking is whether `A` and `B` have the same current state. Let us explore the consequences of different ways of arranging the semantics to produce `true` for this surface syntax. The alternative that comes most immediately to mind is the one posed at the beginning of this section: let names introduced in an **Array** declaration be variables whose values are tuples of a certain form. Give these names the mode $\langle\pi\rangle$, so that when they appear in an un**shared** context, such as operands to =, a `contents` is automatically introduced. This produces the `true` result, but it has several untoward consequences. The first is that whenever an array is passed as an argument to a function, there will be an implicit `contents` operation introduced unless that argument position has a **shared** bindclass. While we do not have an empirical study to back up the claim, it seems probable that in most cases where arrays are used as objects with state, the tendency will be to pass them to functions with the expectation that side effects will occur, so that this proposal would probably cause a proliferation of **shared** bindclasses in function headers, and a strong correlation between an array type and this bindclass. A further problem arises when costs are considered. The `contents` operation on a place containing an `integer` is relatively inexpensive, in both time and space. But a `contents` operation is expensive on large arrays, and it seems inadvisable to implicitly introduce an operation with such great expense. A programmer will necessarily be aware of the expense, and is likely to begin using **shared** not because he wants to modify the array, but to avoid the expense. This leads to a burdening of the **shared** bindlcass with an additional purpose, thus blurring what is to be understood by its appearance.

A second problem with an array as the place of a tuple comes from considering what happens with assignment. If arrays had this semantics, consider the effect of a statement like:

```
A[2, 3] <- 1
```

In kernel terms, this changes the `contents` of the place given by `pselect(A, <2, 3>)`, but it also, in any reasonable semantics, would have to change the `contents` of `A` itself. This violates the axiom ASGN-CNT2, because the place `A` is surely not equal to (using =, of course) the place `pselect(A, <2, 3>)`. To support the semantics of an array as the place of a tuple, we would thus have to have a more complicated version of ASGN-CNT2, by giving some topological property of places, something to the effect that contents don't change so long as the places don't intersect. This is not impossible to do, but recall the ground rules under which axioms must be written. Because the necessity precludes the formulation of axioms using an external model, whatever operation is used in the statement of ASGN-CNT2 must itself be subject to extension, i.e., it must be something whose definition the user can contribute to. This greatly complicates the responsibilty of the user when doing an extension. Further, a more complicated ASGN-CNT2 makes it that much more difficult for analysis tools like common subexpression detection.

The arguments for arrays as places as against arrays as a different kind of place may be summarized as follows. In favor of arrays as places:

`A = B` tells something more interesting than whether two arrays are the same maps to places.

`A <- B` can be used to copy one array into another.

In favor of arrays as they are actually defined:

There are fewer appearances of **shared**, so that it retains a strong association with variables.

An expensive operation is not implicitly introduced.

ASGN-CNT2 remains an equality-based axiom, with attendant advantages of simplicity.

A fundamental tenet in the design of E-L is that the language must support, in a radically improved way, the analysis and transformation of programs. It will, as far as possible, support a wide variety of programming styles, of providing syntax that one might want. But the design of E-L has not been done on a sometimes-might-want basis. Tough decisions are ultimately based on the consequences that they have for formal program manipulation, and in this case, the choice is clear. The structural simplicity of an axiom, in particular, allowing it to be based on equality, overrides the occasional desire to compare the states of arrays using =.

4. Equality

The equality function has had a pervasive influence on the design of E-L, in a way that may not be evident from other documents. The Reference Manual admonishes that equality "is among the first things to think about when you define new types". This principle has been applied relentlessly to the design of many aspects of E-L, not only types, leading to a certain coherence that would otherwise would not exist. This section considers several aspects of E-L whose design has been dictated by thoughts about equality.

4.1 The family itself

Equality is the first example in the Reference Manual of a function family that is treated in detail (section 9.2). It is the archetypal family for several reasons. First, it is an axiomatized family, using the traditional three rules for an equivalence relation. On the one hand, such an axiomatization is a clear practical necessity, if there is to be any analysis at all of a generic function that uses equality. The only possible way to have both extensibility and analyzability is to say that families are governed by rules, which their members must ensure. On the other hand, the rules are based on a classical mathematical notion, that of an equivalence relation. This idea has its origins in the definition of congruence of integers modulo *n*, introduced by Gauss in the opening pages of *Disquisitiones Arithmeticae*, and is used in countless ways in standard mathematics. The mathematical heritage gives us some reason to hope that we will not encounter untoward consequences of equality's strictures, which after all, we have imposed for largely pragmatic reasons.

A second reason for the importance of equality is related to the use of equivalence relations in mathematics: it is occurs specifically in mathematical "constructions", almost invariably with functions that are well-defined over the equivalence relation. Gauss not only defined congruence modulo *n*, but immediately went on to show that addition, subtraction, and multiplication were well-defined over the relation. Thus, the discussion in the reference manual of addition in connection with integers modulo *n* is not without historical precedent. There are, of course, many other mathematical constructions that can be implemented on computers, and many have been, long before the design of E-L. The point is to support the approach, so successful in mathematics, quite directly in the programming environment and language.

This brings us to a third important facet of equality, its role in forming the boundaries of an encapsulation. There is often a choice in the construction used to implement some particular type and associated operations. For example, in the integers modulo *n*, we can keep representatives reduced, or not. Similarly in rational numbers, we can require that the pair of integers representing a rational number be relatively prime, or not. The different choices of representations and the different implementations of fundamental operations on the representations often have different costs, and it is not always clear a priori, or even constant from application to application, which of the choices has lower costs. Nevertheless, if an application uses only functions that are well-defined on the type (including its equality function), then the implementation can be changed, affecting only the cost. Again, this is not a point new to E-L, but the availability of an equality function, and program analysis and manipulation in terms of well-defined functions, may encourage programmers to use these old ideas more effectively.

4.2 Conversion

To implicitly convert or not? That is a question that has often plagued language designers. It is clearly possible to allow too much conversion. The disastrous and unintended complications of all the conversion allowed in PL/I are generally conceded. But completely disallowing implicit conversion is also acknowledged to be a great pain; programmers working in such languages tend to complain and/or cheat. How should E-L come down on this question?

In answering this question, we were fortunately governed by a fundamental E-L design priniciple that whatever is done for built-in types must be available for the user's types. This

caused us not to focus on what might or might not be intuitively reasonable for types that we understood, like `integer` and `boolean`, but to look for principles to apply. With equality always in the front of our minds, it was natural to ask what equality might say about the situation, and this question quickly leads to the relationship between = and `convert` that is stated when conversion is first discussed (Reference Manual, section 10.2): conversion supplies the criterion for cross-type equality. But in truth, it was equality that supplied the criteria for conversion.

Basing conversion on considerations of equality allows some of the conversions that have served programmmers relatively well, for example, from `integer` to `real`. It also disallows some conversions that have not served very well, for example between `boolean` and `integer`. There is a surprise, however, in the partial conversions that are required by the symmetry of equality. To our knowledge, only Common LISP has any trace of this idea, in its partial conversion from real to integer. One would be curious to know *their* rationale. There is no doubt that an argument can be made that sometimes one might want an implicit conversion not based on equality. But E-L, to the best of our ability, has not been based on a sometimes-might-want approach to language design. For many reasons, the goal is a principled design, with few principles, each having maximum consequences.

One of the nicer consequences of basing implicit conversion on equality concerns function application, a point at which some of that implicit conversion occurs. If a function is declared with types for its formals, there will be implicit conversion done by the mechanism responsible for applying typed or templated functions. Because of the relationship of equality and convert, the values of the actual parameters and their corresponding formals will be equal, i.e., =, even if the types change. Thus, analyzers that trace equality information, and this is a large class, can continue to provide useful information, even when they cannot determine exactly what the types are.

4.3 Assignment

Let us consider the possibility of extending assignment, for the moment forgetting what is in the Reference Manual, chapter 12. What might such an extension mean? After the experience with equality and conversion, it is natural to try to discover reasonable axioms of assignment. Attempting to stay away from any preconceptions, an intuitive explanation of assignment is:

> The value of the right hand side is associated with the left hand side, so that when the left hand side is mentioned at some later point, the value can be retrieved. But the association does not last forever, because it can be supplanted by subsequent assignment to the left hand side.

Now, try to bring equality to bear on the problem: where in the above description is there anything about equality? We claim there are two instances: in the phrase "the value can be retrieved", what is meant is the *same* value as the original right-hand side. Similarly, the phrase "subsequent assignment to the left hand side" refers to the *same* left hand side as the original left hand side. Given our predilection for basing semantics on equality, we formalize "same" using =.

If we apply this principle to most programming languages, we get into trouble. Consider the consecutive assignment statements A <- C; B <- C. When we say "same" left hand sides, we cannot ask whether A = B, at least not with the same equality as used in the program. We are left with the choice of proliferating notions of equality, or revising our notion of assignment. The former approach was evidently taken in Common LISP, which has =, eq, eql, equal, char=, char-equal, ... For the design goals of E-L, the latter approach is preferable, and is the reason that the fundamental assignment function is `assign`, rather than <-. By accepting the primacy of equality, one is lead to the notion of "place" (this is what a left hand side is), and the emergence into visibility of the usually implicit `contents` function. These lead naturally to extensions that are otherwise inaccessible (the sparse array example), and to a closer correspondence between analysis and execution (for example, tracing aliases amounts to tracing place values). Again, focusing on equality leads the design in what we consider are desirable directions.

4.4 Types

Chapter 1 of the Definition discusses the question of equality of types in great detail, including rationale issues, and it is not the point to review all of that material here. Very briefly, each elaboration of a type declaration introduces a distinct type, one that is not equal, i.e., #, to other types. The reason for the disequality is that types have functions associated with them, and in separate elaborations of a single declaration, one cannot guarantee that the functions are the same. In order for operations on types to be well-defined, in particular those operations that locate functions associated with the type and apply them, it is necessary that the types be different.

Similarly, in thinking about types that are the results of type constructor applications, these should be distinct from atomic types and from types that are the results of applying different type constructors. Because one of the operations on a type is retrieving the values that were arguments to the type constructor, for the retrieval operations to be well-defined, one wants distinct types resulting from application of a type constructor to different arguments. Note that this involves the equality of other types in the equality of types themselves.

The above issues are all quite simple and non-controversial. The only interesting question is, how should the equality of cyclic types be defined? Section 1.6 of the Definition has a somewhat intricate plausibility argument that a reasonable definition for cyclic types is given by a unique fixed point rule. For a type *t* and a template *tpl* other than the identity:

t = `cyclic(`*tpl*`)` if and only if *t* = *tpl*(*t*)

It is not even immediately apparent that this is a definition, but as things turn out, there is one and only one way to define equality obeying this rule, and as explained in section 1.6, there is a beautiful connection with regular languages that may be exploited, generalizing the more ordinary tree-based techniques for template matching. In presenting cyclic modes to the first time reader, it is reasonable to use the plausibility argument of section 1.6 as the primary rationale for the definition of cyclic types. But for those steeped in considerations of equality, there is at least as much appeal in the unique fixed point axiom and its relation to regular languages. In any case, the definition of the equality of cyclic types has a gratifying inevitability to it.

5. Flow of Control

Our emphasis in providing flow of control constructs in E-L, as in many other areas, has been on providing a mechanism whereby the user can contribute to the extension process. Several of the built-in constructs are standard, and not at all controversial, like **`If then else`** and **`While do`**, and are not discussed here. We do, however, review some of E-L's novelties.

5.1 Iteration

E-L takes a very different approach to iteration from that found in most languages—there is a single syntactic construct for iteration over a sequence of values (actually, over a sequence of value multiples). Even iteration over integers uses this general mechanism, and the user is encouraged to provide iterators by built-in syntactic help, such as **`Iterator`** declarations, and by a tie-in to types, via the `iterate` family.

It is unfortunate that most language designs short-change the user by not allowing user-defined iteration. The lack becomes more glaring as the level of the language gets higher. Many languages [Ada, Pascal, Lisp] allow a great deal of data abstraction, but provide no decent mechanism for stepping through the elements of an otherwise nicely encapsulated data type, although CLU and Alphard provide some support for doing so. While we expect that heavy use will be made of the natural connection provided between types and iteration, we should emphasize that the connection is actually made in two stages. First there is the syntax—**`For in do`**—which sets up an application of the function `iterate`, and then there is the extension of this family by types. The user can exploit the syntax without extending a type, by using the convenience of **`Iterator`** declarations, and this is utilized in several built-in iterators, for example, `to`, `fold`, and the `[...]` notation.

The mechanism underlying the syntax and extensions is quite simple. As discussed in D5.2, the body of an iteration literally becomes the body of a lambda expression, where the names that receive the iterated values are the formals. The reductions are slightly counter-intuitive at first, because what one might think of as "results" of iteration are in actuality arguments to this lambda expression. However, it is some respects amazing that the conncection between iteration and function application has not been exploited more heavily. After all, iteration binds differing values to a name (or names), and executes the same piece of code. This is the hallmark of a function.

5.2 Processes

The idea of coroutines was introduced nearly thirty years ago in the classic paper [Morris]. However, the ideas have seldom been incorporated in popular programming languages, perhaps because they have stack-based implementations and consequently simple storage management languages.

In the design of E-L, there has been no doubting the necessity of supporting a coroutine style of programming. The expressiveness, and in particular the modularity, that it allows is crucial when it is needed. A classic example is that of merge (see RM11.2), but there are many other cases in which a program written for a call-and-return evaluator will be terribly intricate and involuted, only because there is no good way to separate sequential access to data structures from the flow of control of the program. Writing such programs with processes makes one realize that the alternative version of the program has been turned "inside-out", only to get the flow of control aspects working right.

There are a few languages, like Scheme, that allow flow of control patterns that are essentially coroutine-like in nature, based upon the idea of "continuations", which in the absence of optimization can lead to considerable runtime overhead. Scheme has quite an elaborate storage management system already, so the incremental cost of using continuations may not be an important consideration in its design. While recognizing the necessity of supporting coroutine flow of control, we nevertheless have not wanted to commit E-L to an overly elaborate storage management regime for every program. Thus, rather than taking the notion of continuation as

primitive, we have provided the primitive routines described in D5.3. Only one of these (`new-process`) interacts with storage management. As seen from the examples, the use of these primitives seems quite natural, at least as much so as continuations. Save for storage management issues, their implementations are quite simple.

6. Function Families

A *polymorphic* function is one that can be applied to any of several different kinds of data to produce the same or a related abstract effect for each. In most languages, the arithmetic operators are polymorphic; + can be applied to integers to yield an integer, to floating point numbers to yield a floating point number, and so on. The ability to program polymorphically lends greatly to the readability and re-usability of programs. Without it, it is often necessary to write several versions of what is essentially a single function, simply to cater to the different possible data types of its inputs. For example, a `sort` routine might have to be replicated in order to work for arrays of different types, simply because a single function could not legally accept differently typed arrays or use the same comparison operator for different types of elements.

On the other hand, some languages have achieved polymorphism by giving up static type declaration altogether [PPL]. This diminishes clarity and analyzability. EL1 [ECL] permits a mixture of static checkability and polymorphic flexibility; function parameters can be declared to have one of a finite set of types (or else the infinite set `ANY`, which covers any actual argument type), and types are runtime values, so that conditional code for different argument types can be included in a single function. However, EL1's approach has two drawbacks of its own. One is that it is not easy to declare an infinite set of admissible formal parameter types and still retain some selectivity. (The `sort` function might like to accept only `array`'s, but of any element type.) This problem is handled in E-L using type templates (see next section). The second problem stems from the desire to make incremental changes to a program incrementally. In EL1, when a new type is defined for which it makes sense to extend a polymorphic operator, it is necessary to edit the previous version to expand its formal parameter type and add conditional code to cover the new case. It would be preferable to be able to make a completely separate declaration to cover the new case without even touching the previously written and debugged code.

Smalltalk and other object-oriented programming languages [Smalltalk, Trellis] make ease of incremental program adaptation a primary objective. In Smalltalk, when one defines a new class of object, one can define the "method", i.e., function definition, to be used when a particular function name is applied to instances of that class. The methods corresponding to a given name comprise a *family* of functions that achieve a presumably uniform effect on different types of data. This approach is quite successful for functions that are primarily unary, i.e., where the interesting polymorphic behavior occurs in only one argument position. The trouble is, the world of functions is not all unary. For example, `print(object, output-stream)` represents a function family that might reasonably be extended both to new types of objects and new types of output streams. Smalltalk and similar languages force the choice of one or the other.

Ada's "overloading" feature [Ada] is a more symmetric approach to the incremental adaptation problem. Overloading allows the programmer to associate different function bodies with a given function name for use in different calling situations, as determined by the number and types of the actual arguments and the expected result type at a given call. Instead of behavior being associated with individual types, it can be associated with type combinations. Neither functional abstraction nor type abstraction dominates; both are supported and the connection between them is a natural one.

E-L's function family mechanism is similar in some respects to Ada's overloading facility; in particular, it is a clean solution to the incremental adaptation problem. In E-L, a family is a mapping from a tuple of values, called the *parameters* of the family, to a function body that specifies corresponding behavior. As in Ada, the intent is that the arguments to a call on the family will determine the parameter values from from which the particular family member to call can in turn be found and called. But there the similarities end. Apart from some informal rules of good style that govern some of the built-in operators, Ada has no notion of the declaration of a function family as such. A "family" comes into being as the result of a series of individual function declarations with the same function name. In E-L, a family is a declared entity, with clearly stated stucture and rules of behavior. It may not be extended in an arbitrary way. For example, in E-L any extension of the + operator must conform to the rules given in RM10.4. It must be pure, well-defined, commutative and associative. It must take two un**shared** arguments and return

one **unshared** result, all of the same type. Now, many families may have less rigid structure than the + family. But in every case, the programmer is expected to describe the rules that family extensions must conform to. In E-L, the parameters that are the keys to the function family mapping are not simply extracted in some predefined way from the header of the family member. The number and types of a family's parameters are in general declared when the family is created, and their relationship to the formals of the family members can be different for different families. Although + is a binary family, it has a single parameter, namely the type that is the type of a given family member's arguments and its result. Of course, E-L makes it easy to declare standard families in which the rule for taking parameter descriptions from a family member's definition is as simple as Ada's. However, the ability to tailor a particular family's behavior is extraordinarily powerful. The `convert` family, discussed at length in chapter 3 of the E-L Definition, is a good illustration of how complex rules of family behavior can be constructed without having to "hard-wire" them into the primitive semantics of the language.

Ada's overloaded functions act like extended, structured names. It is as if the signature of the overloading function body were appended to the function name to form a compound identifier. Family extensions in Ada obey the same scope rule as if they were individually named, and the language evaluator in effect generates a compound name from a call on the family using the argument and result descriptions and then looks up that name as it would any simple identifier. This is not at all the E-L view of function families. In E-L, a family has a value in its own right, namely the mapping from parameter tuples to member functions. The purpose of an `Extend` declaration is to locally augment that mapping, not to introduce a compound identifier into the current lexical context. This design stems in part from E-L's type flexibility. Whereas Ada requires strict typing of formal parameters and results, it is very useful in E-L to be able to extend a family for an infinite class of parameter values at once, as in

```
Extend + for modulo(?n) with ...
```

so that it is impractical to think of the family member's signature as a hidden part of its name. Furthermore, since a function like `sort` may have a generic type description such as `array(< ?length >, ?elm-type)`, it is not practical for family member declarations to follow the same lexical scope rules as simple named constants and variables. To see why, imagine that the `sort` function has been defined as shown here in skeleton form:

```
Function sort(a:array(< ?length >, < ? elm-type >)
    . . .
      If a[i] gt a[j] then  . . .
    . . .
```

Now `gt` is a function family. The particular member that is chosen when `sort` is invoked will depend on the element type of `sort`'s array argument. Conventional languages with strong static typing and a lexical scope rule would prohibit the use of `gt` within `sort` unless it were declared in the lexical context enclosing `sort` and visible at its point of declaration. E-L requires the *family* `gt` to be declared and visible, but allows `sort` to make use of family extensions made after `sort`'s declaration. In particular, if a new type `t` is declared after `sort`, `gt` can be extended for use on `t` values, and `sort` can be applied to an `array(< ... >, t)` without error.

E-L tempers this flexibilty with axioms of family extension that discourage code that is deceptive or inconsistent. For example, the behavior associated with a type via function family extension must be introduced at the time the type itself is introduced, not delayed so that the type exhibits one behavior for part of its lifetime and another for the rest. This rule is especially important for the `convert` family. Suppose `f` is a function that involves two types, `t1` and `t2`, and suppose that at some point `f` attempts to convert a `t1` value to type `t2`. It would be unacceptable for the result of this attempt to depend on the dynamic state of the `convert` family if that state could be changed

at arbitrary times in the lives of *t1* and *t2*. In such a case, two apparently identical calls on `f` could have drastically different effects. Therefore, E-L's laws rule out extensions to `convert` that would change the behavior of any previously defined type.

7. Type templates

As mentioned in the previous section, the design of E-L stresses the importance of polymorphic programming. The function family mechanism is one novel feature that results from this emphasis. A related but independently useful feature that also supports polymorphism is the type template and the ability to use ? formals in declarations and to have corresponding implicit parameters added to the local scope. For example, consider a function that returns the sum of the elements of its array argument:

```
Function array-sum(a:array(< ?length >, ?elm-type))
                    -> (?elm-type)
  Variable total:elm-type is 0;
  For i in 1 to length do
    total <- total + a[i];
  total
```

Here the ? formals `length` and `elm-type` are bound as additional formal parameters of `sum`, and their values are extracted from the actual argument in the course of conversion.

To our knowledge, no implemented language has included a mechanism this general for the description of generic types and the extraction of implicit parameter bindings by matching with templates. Alphard [Shaw] has a similar facility, however, and although it was never implemented, its designers made extensive studies of example programs in the language. They were enthusiastic about the expressive power of Alphard's template-like feature. It is intuitively clear that templates support polymorphic flexibility while retaining excellent readability and declarative precision.

On the other hand, the study of conversion given in chapter 3 of the E-L Definition makes it entirely understandable that such a feature has not occurred in other programming languages. It is somewhat unsettling that the apply rule for the `convert` family cannot be guaranteed to terminate when type templates are used instead of plain types as conversion targets. (This apply rule is the algorithm that chooses a conversion path from the type of the source value to the desired target type.) Nevertheless, we hold that the usefulness of the template facility outweighs the possibility that evaluation of some conversions might not terminate, provided such cases are extremely unusual. We believe that our conversion algorithm will in fact handle nearly all cases that arise naturally in practice. And of course, it will not actually run forever; it will stop and warn the user if it seems to be looping fruitlessly. We decided that the risk of occasionally having to interact in this way with the environment is preferable to the imposition of artificial strictures on the use of templates. Experience may reveal a set of contraints that can assure termination without fettering the programmer unnaturally. Meanwhile, we expect little or no inconvenience to result from the undecidability of conversion with templates.

As noted in D3.6, one pleasant result of the design of the `convert` family is that a potentially troublesome problem arising from the use of type templates turns out not to introduce additional undecidability and in fact has a nice solution. The issue arises during conversion when a target type template depends on ? formals that the source type template doesn't mention. For instance, consider

```
Extend convert for mixed-sequence, array(?d, ?t) with ...
```

Here we're imagining that mixed-sequence is a `compound-place` of arbitrary length whose elements need not have the same type. It is natural to expect to convert a particular `mixed-sequence`, all of whose elements happen to have the same type, to an appropriate `array` type using the `convert` family member declared in this way. And if a specific target type, such as `array(< 42 >, integer)`, is in hand, it is clear how to obtain a binding for the array dimensions parameter `d` and the element type parameter `t` by matching against that specific target type. More interesting is what happens when this particular conversion rule is part of a series

of conversion steps that may lead on to other types from which no binding for one or both of `d` and `t` can be gleaned. That is, the ultimate target might be a template `sequence(?t)`, representing homogeneous linear sequences of arbitrary length, that is reachable by the rule

```
Extend convert for array(?d, ?t), sequence(?t) with ...
```

In this case, what can the `?` formal `d` be bound to when the earlier `convert` family member is called? The answer, given in chapter 3 of the Definition, is that `d` can be bound either to an actual dimension tuple or to a value of type `template`. The family member then simply does a dispatch of the form

```
Type-case d of
   ntuple(integer) => (there is an explicit value for d);
   otherwise (the result need not satisfy any constraints on d)
```

When array dimensions are not explicitly specified by the target type, the conversion function is free to create an array with whatever dimensions are suitable given the source value.

The `convert` apply rule has the freedom to choose which path to take from a source value type to the target because the axioms of the convert family have been carefully chosen to make sure that it makes no semantic difference which path is taken. For instance, in the example above, suppose there were some other type template *T* through which an `mixed-sequence(?t)` might convert to `sequence(?t)`. Then, since the axioms for `convert` require *T* to be interconvertible with `array(?d, ?t)` either path is equivalent to the other.

The `convert` family is an excellent paradigm for family specification because it shows how axioms for family members can be designed to ensure effective operation of the family dispatch method. For example, the apply rule for `convert` depends delicately on axiom CV3' (section D3.2), which specifies what each family member can expect to be given and what it is expected to produce.

8. Syntax

8.1 Grammatical conventions

The syntax of E-L is novel in several respects. By comparison with most serious languages for systems implementation, it is relatively free of context delimiters, such as **begin** and **end**. Instead it makes use of two-dimensional layout to delimit scopes. Line breaks and indentation carry higher grammatical significance than in most languages, in which they merely separate lexemes. The result is that very few lines are spent on isolated keywords that serve only to bound groups of phrases. The reason for this design is not to save a few extra characters per phrase when a program is input. Tools for structured code entry can handle keywords for the programmer with a minimum of typing effort. Rather, the assumption is that programs will most frequently be read on a video screen. Screen real estate is precious, so there is a premium on keeping program fragments as small as possible consistent with readability. The fact that indentation is significant means that the reader of code is less likely to be misled by a program that is confusingly laid out by its author. What you see is what you get, at least as far as nesting is concerned. Very long program fragments may be difficult to scan with the unaided eye, but it will be simple to provide aids for judging alignment, such as a vertical grid. Furthermore, as the chapters on program preparation in the Reference Manual indicate, E-L will provide numerous inducements to keep code fragments small.

Another fairly novel decision is the use of boldface type as syntactically distinct from plain. Where no confusion is likely to result, we have taken advantage of the distinction to make double use of some important words, such as `type` (**Type**) and `function` (**Function**). This decision is based on the assumption that E-L will be implemented on equipment for which the display and printing of text containing boldface type will be no problem.

The simple grammatical scheme outlined in section 1.3 of the Reference Manual is obviously designed more for ease of extensibilty by users than for efficency of parsing. In E-L, the former is likely to be far more significant than the former, since extensibility is one of our highest design priorities, and since parsing is likely to take place mostly in short bursts interspersed with interactive program entry. Since the software database will usually hold a parsed version of each piece of code it contains there will be little need for large batches of parsing.

Very little of our design time has been spent investigating syntactic schemes for E-L. We have proposed some unusual conventions, and we plan to try them out. If they prove unsatisfactory, we will be able to revert to more traditional methods without severely affecting other aspects of the design.

8.2 Syntactic reductions

The E-L Definition makes abundantly plain that E-L is designed to be reducible in a purely syntactic way to a very small kernel language. Even if such a reduction is not used as the basis for an implementation, the decision to design the language so that it can be will benefits for the user. For one thing, static reducibiity imposes a certain discipline on programmers. As mentioned in section 3, one aspect of the reduction is to eliminate bindclasses such as **shared** and **delayed** and to modify programs to handle explicit place values and explicit functions instead. In order for this to be possible, functional parameters and function results must be carefully specified if they are going to make use of these features. (Chapter 4 of the Definition discusses the calculus of "modes", i.e. syntactic descriptors, used to propagate these specifications.) That means that readers of unfamiliar code will be alerted to uses of these potentially tricky constructs.

A further benefit to E-L programmers lies in the existence of the E-L Definition itself. It will be a formal specification of a substantial portion of the language that—unlike make formal language definitions—can be used by ordinary programmers.

If the syntactic reduction described in the Definition is given a key role in the implementation, as seems quite likely, then additional benefits will accrue. First, the reduction scheme can be made the basis of tools for notational extension. If a large fraction of the language is in fact implemented by

extension, then its extension facilities are likely to robust enough to handle user extensions as well. And users will have even more reason to appreciate the Definition as a guide to the language.

Furthermore, if programs in concrete syntax are reduced before being given to tools for analysis, transformation, interpretation and compilation, these tools can take advantage of the normalization that results when the numerous convenience features of the concrete language are eliminated in favor of kernel forms. In a system with as many different tools for processing programs as E-L will have, the existence of a simplified common representation will have a profound effect on system performance. Of course, the programmer's original input will not simply be set aside once it has been read and reduced by the system. A major objective of the implementation is for the system to interact with the user in his own terms to the greatest extent possible, rather than force him to adjust to program text that has been reduced and reconstituted.

9. Abstract Types

The principle of separating the specification of a type from its implementation is now widely supported from a methodological standpoint, and some language designs, notably Alphard's [Shaw], have gone a long way towards making this principle easy to follow in practice. Still, every language of which we are aware assumes a one-to-one correspondence between a type specification and its realization. The fact is, however, that in large systems development, one often needs several simultaneous realizations for a single type specification, either because each is best suited to some particular task within the system or because the implementor is making a transformation from an initial realization to a more refined one, and it is necessary for both to exist simultaneously while that transformation is in progress.

In E-L, one can specify a type abstractly and postpone connecting a representation to it. Properly specified abstract types can be used in programs wherever types are expected, and tools for analysis and transformation can process these programs in spite of the lack of any concrete representation. Of course, in order to execute such a program, the implementor will supply one or more *realizations* for its abstract types. A realization is a type that has a specific representation and that meets the specifications for the abstract type that it realizes. For example, a programmer might define an abstract type `multiset` that represents integer sets allowing repetition. He could provide functions `insert` and `remove` for inserting and removing `multiset` elements and a function `count(n, s)` that returns the number of elements equal to `n` in `multiset s`. Other functions on `multiset` values might be written entirely in terms of these three; for example, a predicate to test membership in a `multiset` could be

```
Function member(n, s:multiset) count(n, s) gt 0;
```

But having no `Based-on` subdeclaration, the type `multiset` has no representation.

Programs involving abstract types like `multiset` can be understood by human readers and analyzed by tools. In particular, there can be tools that suggest representations for `multiset`, based on the particular programs in which it is used. Suppose that, just to begin debugging, the programmer chooses a naive representation which is a simple list of the elements of the multiset, including duplicates. He creates a type called `simple-list-multiset`. This type has a subdeclaration such as:

```
Based-on cyclic(int-list)
            union({nil}, record(element:integer,
                                rest:int-list));
```

And it also declares itself to be a realization of `multiset` using the subdeclaration

```
Realizes multiset;
```

`simple-list-multiset` must provide definitions for `insert`, `remove`, and `count`, which were only "promised" in the abstract type. It need not supply a definition of `member`, since the definition given with `multiset` is adequate. On the other hand, because the list representation is *so* naive, the programmer may want to supply a more efficient version of `member` with `simple-list-multiset`, a version that doesn't depend on `count`. This he is free to do.

A program written in terms of `multiset` can run with the abstract type bound to a realization such as `simple-list-multiset`. This is how debugging will typically begin. There may soon be other realizations, and in that case, the abstract type `multiset` can be bound to a union of the realization types, or to an expandable union that is capable of encompassing representational variants as they may arise in isolated regions of the program. Ultimately, the implementor will

wish to decide—perhaps with the help of tools—which specific occurrences of `multiset` should continue to act like unions of several concrete types, which should be treated like `one-of` type designations, and which ought to be bound to specific realizing types. These decisions can be implemented using the program refinement machinery of E-L. The result will be to annotate selected scopes of the program or even selected occurrences of the abstract type, so that an appropriately precise and efficient realization is used in each case.

10. Object-Oriented Programming

The increasing popularity of object-oriented programming, in environments that are dedicated to that approach [Smalltalk, Trellis], in most dialects of Lisp that are under active development [Flavors, Commonloops, HPLisp, ObjectLisp], and in some implementation language variants [ObjectiveC], naturally invites comparison with the style encouraged by E-L.

Like E-L, Smalltalk is an environment designed in tandem with its programming language and based upon many of the same concerns for productivity, for example in the ease and safety of modifying existing systems. Smalltalk, however, is not intended to emit free-standing production versions of programs. The features of E-L that permit a great deal of control over representation choice are absent from Smalltalk and are at least superficially inconsistent with its basic style. Furthermore, as we have discussed previously, Smalltalk's "unary" bias is overly confining in a number of situations. The E-L function family mechanism seems to achieve the same benefits in a more general and flexible way.

In cases in which object-oriented facilities have been added to existing environments and languages, such as the Flavors subsystem of Zetalisp, the results have been useful and popular, but from a language designer's point of view, they seem baroque and complicated to document and to learn. Apparently, it is difficult to blend such different programming idioms in an elegant and simple way.

As we made clear in section 3 on the concept of state in E-L programs, we are entirely in sympathy with the view that is at the basis of object-oriented programming, namely that there are many programming problems in which the most appropriate way to model the key entities being manipulated is as values with state, i.e., objects. We further believe that other aspects of the object-oriented style have beneficial effects on productivity and reliability in software development. For example, the ability to derive one type from another incrementally, by inheriting its properties and then augmenting it with additional state and behavior, is an excellent way to re-use proven material without having to alter it. The generalization to multiple inheritance, which permits the composition of several existing types, or collections of useful behavior that can contribute to types, is also extremely useful. But because so many issues are raised by the attempt to combine types gracefully (how to resolve conflicting component and function names, what type to ascribe to the parameters of inherited functions, how to access obscured elements of inherited types, or "superclasses" in the jargon of object-oriented programming, and so on), we feel that object-oriented capabilites should be achieved by extension, rather than building them into the language.

E-L has the flexibility to allow this to be done. The expandable union type can be used to represent a class of object instances having certain components and behavior. New subclasses can be declared as variants of these classes. Multiple inheritance, or type composition, presents no difficulty since a given type can be a variant of more than one class, i.e., expandable union. Because the mechanism of function family dispatch can be controlled completely, it is possible to define a category of function families that might be called "methods", using the Smalltalk term. Methods can have rules of dispatch that search the class hierarchy to find function attributes as would an object-oriented evaluator. Composite types can be obtained by embedding the constituents in a compound representation and then hiding the extra level of structure with an extension to the appropriate selection function.

In short, E-L can be extended by its users or by library package designers to gain the benefits of object-oriented programming without compromising its design principles, including reducibility to a simple underlying mechanism.

END

DTIC

8-86